

PROGRAMMING EXPERT

FAQ

Q What is eXtreme Programming (XP)?

A According to traditional programming methods, software should be treated like an engineering project. It should be planned in detail, then the plan should be executed perfectly. When building a bridge, this is the only sane approach; the cost of changing your mind from a box girder to a suspension bridge is high. But when it comes to software, change is cheap.

XP is a set of guidelines that tries to use the ease of changing software to its advantage, hence its alternative name 'agile programming'. XP introduces refactoring – changing class structures mid-project – and pair programming – having two programmers working on the same lines of code so neither owns it. XP demonstrates that adapting and changing software can make it better, not buggier.

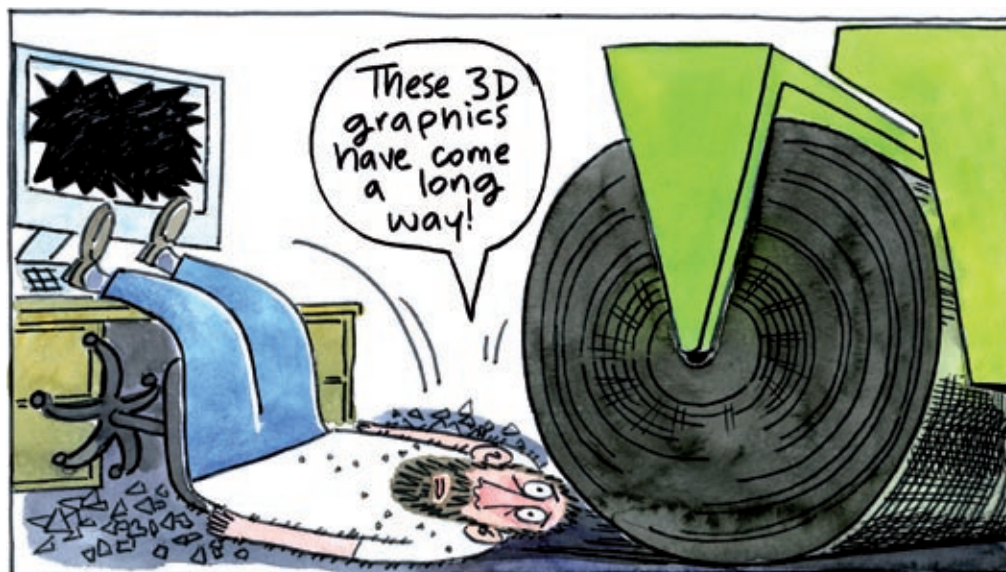
Mike James

IT author, lecturer
and programmer

mike@computershopper.co.uk

Using 3D software with .NET

If you're a .NET programmer in need of 3D tools, the graphics facilities in the Windows Presentation Foundation environment can help broaden your horizons, as Mike James explains



Whether you want to create an action game or polished business charts, 3D graphics are essential. Until recently, however, the .NET programmer was short of 3D tools. Microsoft had dropped development of the managed extensions to DirectX and it seemed the only way to work in 3D was to use the XNA development studio (*Programming Expert*, *Shopper* January 2007), which is targeted heavily at Xbox development. The next generation of the .NET system has come to the rescue. The .NET 3.5 beta uses a new graphics system called the Windows Presentation Foundation (WPF). It introduces a new way of creating form-based applications. We took a look at the WPF form controls and the XAML markup language in *Shopper* February 2007. This month, we'll look at a different aspect of WPF: 3D graphics.

Classic Windows forms are rendered for display on a physical device by the Graphics Device Interface (GDI) and, more recently, by the GDI+. XAML forms are rendered by the DirectX 9 graphics engine and don't use the GDI at all. Many of the WPF graphics facilities are managed wrappers and extensions of DirectX 9. This means all drawing is 3D and, with the correct graphics card, hardware-accelerated. This causes difficulties if you try to mix old-style GDI/GDI+ graphics with the new WPF graphics, but any problems are outweighed by the benefits of the sophisticated and fast DirectX graphics system.

Things are still not perfect with WPF. It uses the DirectX 9 graphics engine, but the latest version is DirectX 10 and at the moment there is no managed class library for DirectX 10. If you want the best possible graphics you need to use DirectX 10, but it's available for Vista using only C or C++. The XNA Studio is also a managed DirectX 9 system. Given the advantages of the WPF (not least of which is that it's a

complete 3D framework) and its backward compatibility with Windows XP, it seems a good choice.

To use WPF you must either install the new .NET Framework or, more simply, download and install the latest Orcas versions of the Express Editions of the Visual programming languages. The Orcas versions provide full support for WPF and automatically install the .NET 3.5 Framework. They also have a drag-and-drop editor, which can be used to create WPF forms without having to get involved in XAML or write much code. The Express Editions of Visual Basic, C#, C++ and Web Developer can be downloaded from <http://msdn.microsoft.com/vstudio/express/future>. For this project we'll be using the C# version, but everything works with just slight changes in Visual Basic and C++.

There is also a full version of Visual Studio Orcas available for download, but this is probably overkill for trying out WPF. All the Orcas versions are at beta 1. In



▲ The Orcas webpage, waiting for you to select the version you want to download

practice they seem stable and usable, but the usual warnings about not installing beta software on an important machine apply. In addition, the licence agreement states that you can use the betas but not to produce commercial releases.

You don't have to uninstall any existing Express Edition products you might have, as the new Orcas version will happily coexist with them; simply select the one you want to use for a given project. If you open the Orcas version, Microsoft Visual C# 9.0, you'll see the only big difference is that now, when you start a new project, you are given a choice of Windows Forms: WPF Application and WPF Browser Application. To create a traditional GDI/GDI+ application, select Windows Forms. A WPF application uses the DirectX 9 graphics engine and the new XAML-based forms and controls. The Browser Application uses XAML as a markup language and the new Silverlight browser add-in. At the moment, it isn't clear if Silverlight will become an accepted web technology and it certainly makes things more confusing by adding another way of creating web applications.

3D FUNDAMENTALS

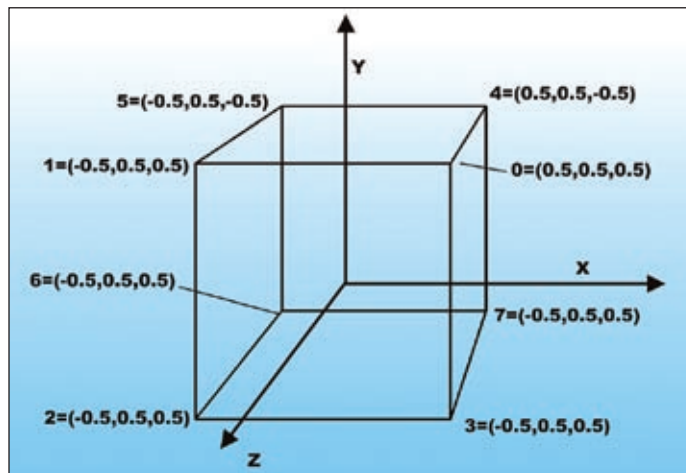
In *Programming Expert*, *Shopper* February 2007 we looked at using WPF for general-purpose forms with 2D buttons and so on, all rendered behind the scenes by the DirectX 9 graphics engine. This project starts from the other end of the spectrum and looks directly at the 3D graphics facilities. This gives a unique perspective on the framework and will help you see how to use forms and 3D together.

With WPF, you create a 3D model and define an associated view to produce a 2D bitmap for the display. You use the `Viewport3D` object to take the 3D model and render it as a 2D visual element that can be placed in a container such as a grid or a canvas. In this sense, the `Viewport3D` is just another control (like a button or a text box) that can be placed on a form. This is how 3D and forms fit together.

If you are familiar with DirectX 9, you will recognise many 3D objects in WPF. However, you would need to go back to before DirectX 8 to have encountered one particular feature. WPF uses retained mode graphics, which are ideal for forms and buttons. If you draw a button on a form and another form temporarily covers it up, the system takes care of redrawing the button when it becomes visible again. That is, all the graphics you place on a form persist until you explicitly dispose of them. This is ideal for forms, but it's not the way DirectX 9 normally works. With moving 3D graphics, it is faster and more efficient for the game to work out which parts of its 3D world need to be redrawn.

DirectX 9 has no native retained mode, but WPF gives it one. This lets you set up a 3D model just once and then allow the system to render it as required. You produce animation effects in a retained system by moving and rotating objects in the model and/or moving the viewing position. Retained 3D is ideal when you want to create a model of a city that you are going to fly over.

Another big difference between WPF and DirectX is the availability in XAML of both a descriptive and a procedural way of creating models and views. XAML is like a reinvented HTML and can be used to specify user interfaces and graphics in much the same way. The big difference is that any XAML within your project is converted into objects with which the code of



◀ The cube and the coordinate system

your project can interact. For example, you could create a button using XAML like this:

```
<Button Name="button1" >
  Click Me!
</Button>
```

You could also create a button using C# code:

```
Button button1 = new Button();
button1.Content = "Click Me";
```

The result of these two approaches is exactly the same, as the XAML code is read by the application engine and converted into an object called `button1` of type `Button` during initialisation. You can also mix the two styles of working, perhaps creating some graphics objects in XAML and then manipulating them in procedural code. There are many advantages to using XAML, but in our example we'll use a pure procedural approach because this is closer to the way most other 3D systems are used and it keeps the definition and use of an object together.

MAKING A MESH

It's time to start coding. WPF has only a single basic 3D shape or 'primitive': the mesh. A mesh is a collection of 3D points or vertices organised into triangles. Each triangle defines a portion of the surface. In 3D graphics there are no circles, ellipses or curves – just collections of triangles that approximate the smooth surface. In most cases, meshes are created using a graphics editing tool and then translated into numerical definitions. However, it is worth creating one numerical definition by hand, if only to discover why you need a graphics tool to do the job.

Start a new WPF Application called `Cube1` and add:

```
using System.Windows.Media.Media3D;
```

The `Media3D` namespace is where the 3D classes we'll use are found.

To get started, we need a 3D mesh that can be rendered. The simplest useful mesh you can create is a cube positioned at the origin of the xyz coordinate system (x is left-right, y is up-down, and z positive out of the screen, negative into the screen). When you first discover how a mesh is defined, it seems overcomplex. First you need to define an array of all the points making up the mesh. Then define triangles by specifying each three points in the array that make up each triangle. In a continuous surface, the same point

will be used in several triangles. We'll create a function to return a cube, defined as a `MeshGeometry3D` object. It starts:

```
MeshGeometry3D MCube()
{
  MeshGeometry3D cube =
    new MeshGeometry3D();
```

Now we need to create the array of points. In this case, we need eight points corresponding to the eight corners of the cube:

```
Point3DCollection corners = new
  Point3DCollection();
{
  corners.Add(new Point3D
    (0.5, 0.5, 0.5));
  corners.Add(new Point3D
    (-0.5, 0.5, 0.5));
  corners.Add(new Point3D
    (-0.5, -0.5, 0.5));
  corners.Add(new Point3D
    (0.5, -0.5, 0.5));
  corners.Add(new Point3D
    (0.5, 0.5, -0.5));
  corners.Add(new Point3D
    (-0.5, 0.5, -0.5));
  corners.Add(new Point3D
    (-0.5, -0.5, -0.5));
  corners.Add(new Point3D
    (0.5, -0.5, -0.5));
```

Every mesh stores the array of points it uses in its `Positions` collection:

```
cube.Positions = corners;
```

Now we have eight vertices, numbered 0 to 7, we define the triangles that make up the surface of the cube. There are two triangles for each of the six faces of the cube, making 12 in total. The property of the `MeshGeometry3D` object we need to set is called `TriangleIndices`. This is loaded with three vertices for each triangle. We specify the index (a number from 0 to 7 in our case) of each vertex in turn. For our 12 triangles, we'll specify 36 indices in total. Triangle one connects vertices 0, 1 and 2, while triangle two connects vertices 0, 2 and 3, and so on. If you sketch a cube, number the corners and mark off each triangle, you'll find the full 36-element array we need is:

```
Int32[] indices = {
  // front
```

```
0,1,2,␣
0,2,3,␣
//back␣
4,7,6,␣
4,6,5,␣
//Right␣
4,0,3,␣
4,3,7,␣
//Left␣
1,5,6,␣
1,6,2,␣
//Top␣
1,0,4,␣
1,4,5,␣
//Bottom␣
2,6,7,␣
2,7,3,␣
};␣
```

The order in which we list a triangle's vertices determines which way it faces. If you're looking at the front of the triangle, you must list the vertices in anti-clockwise order. Later when we define a view of our object, if only the back face of a triangle is visible then it won't be rendered. This simple scheme avoids rendering the inside surfaces of objects. We must then transfer our list of numbers from an array to a collection (which we'll call `Triangles`):

```
Int32Collection Triangles =␣
    new Int32Collection();␣
foreach (Int32 index in indices)␣
{␣
    Triangles.Add(index);␣
}␣
```

This method of defining an array and then transferring it to a collection involves less typing than adding the index values directly to a collection. Finally, we set the cube's `TriangleIndices` property and return the cube object as the result:

```
cube.TriangleIndices = Triangles;␣
return cube;␣
}␣
```

For a more realistic mesh, we could also define a normals collection and a texture coordinate collection. However, our simple mesh, built only from points and triangles, is all you need to start rendering a 3D image.

RENDERING THE MODEL

Now we have a demonstration mesh we can begin to construct a 3D view. The best place for this code is in the `Window Loaded` event handler. The only problem is that the current beta of Orcas Express doesn't support the automatic creation of event handlers (beta 2 promises to do so). Therefore, we have to add an event handler manually. Add the following line to the `Window's` constructor:

```
this.Loaded +=␣
    new RoutedEventHandler(this.␣
        WindowLoaded);␣
```

Now you can begin the event handler in the usual way:

```
private void WindowLoaded(␣
    object sender,␣
    EventArgs e)␣
{␣
```

Other event handlers can be created in the same way.

A mesh isn't quite a finished 3D model. To be rendered it needs to be associated with a material, which defines its colour and texture. As you are likely to want to create multiple 3D objects with the same mesh but maybe in different colours and textures, it makes sense to use another class that stores a mesh and a material. This is what the `GeometryModel3D` class does, and it is the simplest object that can be rendered.

To create a `GeometryModel3D`, build our mesh and store it in the model's `Geometry` property, three lines are required:

```
GeometryModel3D Cube1 = new␣
    GeometryModel3D();␣
MeshGeometry3D cubeMesh = MCube();␣
Cube1.Geometry = cubeMesh;␣
```

All we need to complete the model is a material:

```
Cube1.Material = new␣
    DiffuseMaterial(␣
        new SolidColorBrush(Colors.Red));␣
```

This makes our cube render as if it were covered in a matt red paint. There are other types of material, and you can apply more complex brushes such as image brushes. The best way to find out how these work is to experiment.

SETTING THE SCENE

Now we have a 3D object, we need to create at least one light and one camera object to view it. One of the simplest types of lights to use is the directional light. This creates light of a specific colour in a given direction. To create a white directional light shining down, to the left and into the scene, you would use:

```
DirectionalLight DirLight1 = new␣
    DirectionalLight();␣
DirLight1.Color = Colors.White;␣
DirLight1.Direction = new␣
    Vector3D(-1, -1, -1);␣
```

You can define multiple lights of different types, but a single directional light is enough to demonstrate how objects change their appearance in response to lighting.

Finally, we need a camera. A perspective camera is the most common one to use because it renders a scene complete with perspective effects, but this does make it slightly more complex to define:

```
PerspectiveCamera Camera1 = new␣
    PerspectiveCamera();␣
```

To set the camera, we need to specify its near and far clipping planes. Any object closer or further away from the camera isn't rendered:

```
Camera1.FarPlaneDistance = 20;␣
Camera1.NearPlaneDistance = 1;␣
```

The camera's field of view is specified as an angle. You can think of this as setting the type of lens the camera is using. A big angle exaggerates perspective and corresponds to a wide-angle lens. A small angle reduces the perspective effect and corresponds to a telephoto lens:

```
Camera1.FieldOfView = 45;␣
```

We'll position the camera at a positive *z* coordinate (in front of the cube) as well as moving it to the right and above the cube (remember the cube is positioned at the origin):

```
Camera1.Position = new Point3D␣
    (2, 2, 3);␣
```

The most difficult setting to get right is the `LookDirection`, the direction in which the camera is pointing. If you get it wrong, the chances of your seeing anything are slim. A handy tip to get started is to place your models at the origin so the `LookDirection` can be set to the opposite of the camera's `Position`. So if the camera has the position *x, y, z*, it should be pointing in the direction *-x, -y, -z*:

```
Camera1.LookDirection = new␣
    Vector3D(-2, -2, -3);␣
```

Finally, the camera needs to be the right way up:

```
Camera1.UpDirection = new␣
    Vector3D(0, 1, 0);␣
```

We are almost ready to render the entire scene, but first we must make sure all the 3D objects that we have created so far – the cube and the light – are gathered together in a single `Model3DGroup` object:

```
Model3DGroup modelGroup =␣
    new Model3DGroup();␣
modelGroup.Children.Add(Cube1);␣
modelGroup.Children.Add(DirLight1);␣
```

In what at first appears to be an unnecessary level in the object hierarchy, we next have to assign the `Model3DGroup` to a `ModelVisual3D`:

```
ModelVisual3D modelsVisual =␣
    new ModelVisual3D();␣
modelsVisual.Content = modelGroup;␣
```

Finally, we create a `Viewport`, which converts everything from 3D to a 2D rendering:

```
Viewport3D myViewport = new␣
    Viewport3D();␣
```

The way in which the conversion is performed is controlled by the camera assigned to the `Viewport`:

```
myViewport.Camera = Camera1;␣
```

We can add the 3D model to the `Viewport` so we have something to render:

```
myViewport.Children.␣
    Add(modelsVisual);␣
```

You must assign the `Viewport` to the container that is going to display the 2D bitmap it creates. A standard WPF application creates a form with a `Grid` layout container as its content. The simplest way of displaying the `Viewport` is to add a `Canvas` layout container to the grid by dragging and dropping from the `Toolbox`. This adds a component called `Canvas1` by default, and we can add the `Viewport` to it using:

```
this.Canvas1.Children.␣
    Add(myViewport);␣
```


We can also specify the 2D physical properties of the Viewport such as its size and location:

```
myViewport.Height = 500;#
myViewport.Width = 500;#
Canvas.SetTop(myViewport, 0);#
Canvas.SetLeft(myViewport, 0);#
this.Width = myViewport.Width;#
this.Height = myViewport.Height;#
```

If you want to use the entire window to display the scene, you could simply use:

```
this.Content = myViewport;#
```

SPIN AND STORYBOARDS

At this point there are many different tracks to explore, but to finish our example let's explore a demonstration of how transformations work. All objects have transformation properties, which can be used to modify where and how they appear in the model. To construct a rotation transformation, we need an axis and an angle to rotate through:

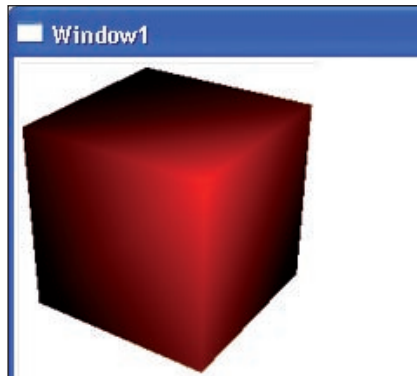
```
AxisAngleRotation3D axis = new#
    AxisAngleRotation3D(#
        new Vector3D(0, 1, 0), 0);#
```

The transform is now created and associated with the cube:

```
RotateTransform3D Rotate = new#
    RotateTransform3D(axis);#
Cube1.Transform = Rotate;#
```

From this point on, we can change the Angle property of axis and the cube will be redrawn at this new angle. The correct way to do this is to use WPF's animation, and as this is very poorly explained in the documentation an example is needed.

Animation is achieved using a Storyboard object, which can contain any number of Animation objects. When the storyboard is



▲ The red cube

activated all the animations it contains are run. You can think of an Animation object as specifying how a property of the object being animated should change over time. First, we need an animation object that will change the Angle property:

```
DoubleAnimation RotAngle=#
    new DoubleAnimation();#
RotAngle.From=0;#
RotAngle.To=360;#
RotAngle.Duration=new Duration#
    (TimeSpan.FromSeconds(20.0));#
RotAngle.RepeatBehavior=#
    RepeatBehavior.Forever;#
```

This defines a repeated animation of a double from an initial value of 0 to a final value of 360 over 20 seconds.

We now have to connect the animation with the object and property being animated. This is slightly complicated in that we first have to register a friendly name of the object we want to animate in the namespace of its container:

```
NameScope.SetNameScope(canvas1#
    , new NameScope());#
canvas1.RegisterName("cubeaxis",#
    axis);#
```

This creates a new NameScope in canvas1 and adds 'cubeaxis' as the friendly name for the axis object. You can think of this as building a list of all the objects we want to animate within a layout container. Now we have the axis object registered, we can specify it and its Angle property as the target of the animation:

```
Storyboard.SetTargetName(#
    RotAngle, "cubeaxis");#
Storyboard.SetTargetProperty#
    (RotAngle,#
        new PropertyPath(#
            AxisAngleRotation3D.Angle#
                Property));#
```

The first instruction sets whatever object has been registered as 'cubeaxis' as the object to animate. The second instruction defines the property on the object that will be changed by the animation.

At last, we can create a Storyboard object and add the animation:

```
Storyboard RotCube=new Storyboard();#
RotCube.Children.Add(RotAngle);#
```

A storyboard can support multiple animations on a range of different objects and properties. Finally, we start the animation to spin our cube:

```
RotCube.Begin(canvas1);#
```

The animation is specific to the canvas1 layout container and only at this point is the name 'cubeaxis' resolved to the real axis object and its property. The advantage of this is that you could apply the storyboard to another layout container with a different set of objects as long as you register the same friendly names.

You can add animations to move the camera's position or its zoom in the same way and create a 'fly past', but to make it look impressive you need more than a single cube. This could be the start of a very big project

ADDISON-WESLEY Doing Objects in Visual Basic 2005



£36



Visual Basic 2005's object-oriented nature is frequently misunderstood or misused by programmers with a VB6 background. This book professes to address the problem. However, author Deborah Kurata seems to want to talk about almost anything but object-oriented programming. It's not until late on in the book that you get to learn about classes and creating them.

A great deal of the first part of the book deals with design methods, database theory and so on, which, while interesting, are not core object-oriented ideas. As a result, this book deals with too many difficult and messy topics and lacks the clear approach to objects that the beginner needs. Parts of the book are readable, but it doesn't live up to the promise of its title and many beginning-to-intermediate VB programmers are going to find it very confusing.

A far better way to introduce objects would be to focus on controls, for which the concepts of properties, methods and events are easily grasped. This is a missed opportunity.

SUMMARY

- Interesting content
- Confusing approach to topic

SUPPLIER www.amazon.co.uk
ISBN 0321320492
PAGES 552

APRESS Accelerated VB 2005



£18.50



Accelerated VB 2005 sounds like a good idea if you want to get into Microsoft's latest object-driven version of Visual Basic quickly and you already know something about object-oriented programming or some other language. The topics covered in Guy Fouché and Trey Nash's book are advanced - threading, exception handling, generics, delegates and so on. They also cover many basic ideas, objects, classes, methods and how to work with strings.

It is advanced but it's also pedestrian, adding little to the standard documentation or what you can easily find on the internet. The authors explain the ideas reasonably well, and the examples are reasonably concise and comprehensible, but at no point do you feel you are being given any special insights into Visual Basic 2005.

The technical level is also a problem: complete beginners will be lost at chapter one, but experienced programmers may be underwhelmed.

SUMMARY

- Clear explanations
- Pedestrian content

SUPPLIER www.amazon.co.uk
ISBN 1590598016
PAGES 414